10/520949

REC'D PCT/PTO 12 JAN 2005
PCT/CA 03/01000

Office de la propriété intellectuelle du Canada

Un organisme d'Industrie Canada

Canadian Intellectual Property Office

An Agency of Industry Canada

11 AUGUST 2003 11·08·03

REC'D 05 SEP 2003

WIPO PCT

Bureau canadien des brevets

Certification

Canadian Patent Office

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification and Drawing, as originally filed, with Application for Patent Serial No: **2,393,502,** on July 15, 2002, by **MARK J. FRAZER,** for "System and Method for Reliable Transport in a Computer Network".

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

[illegible]

Agent certificateur/Certifying Officer

August 11, 2003

Date

Canada

(CIPO 68)
04-09-02

OPIC CIPO

BEST AVAILABLE COPY

ABSTRACT

A system and method for reliably transmitting a request from a client application running on a client system to a server application running on a server system and receiving a reply to the request, in which functions that provide parameters governing the reliable transport mechanism and message forming and parsing services are provided by the application processes to and called from application-layer protocol stacks.

SYSTEM AND METHOD FOR RELIABLE TRANSPORT IN A COMPUTER NETWORK

FIELD OF THE INVENTION

The present invention relates to systems and methods for providing for reliable transport of messages over computer networks. More specifically, the present invention relates to a system and method for providing reliable transport of messages over a computer network that includes an unreliable data link such as a wireless link in a wireless network.

BACKGROUND OF THE INVENTION

To understand the context of the present invention it may be helpful to the reader to have a concrete, but very simple exemplary computer network, such as that shown in Figure 1 and indicated generally by reference numeral 10, to which the reader can refer. The computer network 10 is shown as including two host computers that are referred to here as client host computer 12 and server host computer 14. Those skilled in the art will understand that the designation of a host computer as a client or as a server in this context depends upon which host computer sent a request to begin an exchange of messages between the host computers 12, 14. For the purposes of this discussion it is assumed that the client host computer 12 is running a client-side network application that needs to have some processing done by a server-side network application running on the server host computer 14. Generally, the client host computer 12 will include a central processing unit ("CPU") 16 and memory 18. The CPU 16 runs the operating system 20 of the client host computer 12. In addition, the CPU 16 may run client-side network applications. The client host computer 12 interfaces with a communications network 22 through a communications interface 24. The communications network 22 may be a wired network or a wireless network or a partially wireless network and transports data between the host computers 12, 14. The communications interface 24 may be a modem or other means for interfacing between a computer and communications network 22.

Likewise, the server host computer 14 includes a CPU 26, memory 28 and an operating system 30. CPU 26 runs the operating system 30 of the server host computer 14 as well as server-side network applications. In addition, the server host computer 14 interfaces with the communications network 22 through a communications interface 32.

The elements of a simple computer network described above are known to those of ordinary skill in the art and will not be described further herein.

A client-side network application must rely on the computer network's transport system to communicate requests (sometimes referred to as "remote procedure calls" or "RPCs") to a server-side network application. Typically, a software interface, usually referred to as a "socket", is provided by an operating system 20, 30 for a network application to gain access to the transport system. On the client side, a socket is provided into which a client application can send requests to be transported to the server application. Similarly, on the server side, a socket is provided from which the server application can receive requests sent by the client application and into which the server application can send replies that need to be transported to the client application. Typically, if the computer network's transportation system is based upon the Internet Protocol ("IP"), network applications that need reliable transport use either the Transmission Control Protocol ("TCP") or include their own procedures to provide reliable transport using the User Datagram Protocol ("UDP"). In the latter case, this means that each time a network application is written for or ported to a new network configuration, significant effort is generally required to provide optimal reliable transport.

For example, a client application may need to send requests to a server application through a transport system that includes a wireless link. The transport system may have a high degree of error correction built in and may include a dedicated channel for some requests that the client applications may wish to send to some server applications. In such a situation, the reliable transport mechanisms built into TCP are not optimal for requests transmitted to all serve applications over the wireless link. Because lost requests are very likely to be due to data loss in the wireless link rather than collisions or buffer overflow due to congestion, for urgent requests (e.g., for signaling that a user has picked up a telephone handset in an voice over Internet application) that will pass through the wireless link it may be preferable to retransmit an apparently lost request with a relatively long (computer terms), but constant delay between retransmissions and with a small hard limit on the number of retransmissions before an error is reported. However, for requests that do not pass through the wireless link or are less urgent, but which must be delivered, it may be preferable to have an initially shorter delay between retransmissions, a retransmission delay that increases linearly, and a very large number of retransmissions before an error is reported. In other cases, a protocol that handles retransmission in the manner in which TCP handles retransmissions may be desirable. Moreover, conditions in the network may change with interference in the wireless link or loading of the wire-line portion of the network, even if the network is dedicated to the network application (i.e., is not a public network like the Internet) and the radio spectrum used for the wireless is dedicated to the network.

Implementing a network application that can reliably handle remote procedure calls has in the past been done by either living with the limitations of TCP or building reliable transport into the network application and using UDP. For example, application-layer protocols such as Media Gateway Control Protocol ("MGCP") or Session Initiation Protocol ("SIP") include specifications for reliable transport that can be implemented in network applications that use remote procedure calls for signaling so that the network application itself provides reliable data transport. However, doing this adds additional complexity to the network application and means that the network application must be modified or at least tuned whenever changes occur in the network such that remote procedure calls are transported over paths having new characteristics or whenever remote procedure calls are to be made that have different requirements for timing and reliability. It would be preferable if characteristics of reliable transport mechanism provided for a specific client application that must make remote procedure calls to a specific server application could be changed without modifying the application.

For example, in Voice-Over-Internet-Protocol ("VOIP") applications, TCP and UDP are not optimal if a wireless link is involved. Such applications, if they are to provide carrier-grade voice communication, need to reliably establish a communication channel as soon as possible after a subscriber picks up a telephone handset. While such applications can tolerate some degree of media data loss in the digitized voice signal once a connection is established, signaling messages must be highly reliable if subscriber expectations raised by past experience with the Public Switched Telephone Network ("PSTN") are to be satisfied. Network applications such as VOIP applications are typically implemented in accordance with application-layer protocols such as MGCP and SIP in which reliable transport aspects are intermingled with the application-specific aspects of the protocol. Such applications directly use the services of a transport layer that implements the UDP or possibly the TCP over an IP network.

There is a need for a method and system that provides an application layer that can be adapted or adapt itself to network applications and transport conditions without revising the underlying code in either the network application or the transport layer.

SUMMARY OF THE INVENTION

The inventor, in considering how the task that a network applications programmer faces in writing or porting applications to a new network operating system could be simplified without sacrificing flexibility, concluded it would be preferable to separate as much as possible of the reliable transport mechanism from the network application while retaining UDP as an underlying transport layer. One way to accomplish simplification for the programmer is that typified by TCP; in TCP the reliable transport mechanism is built into the transport layer with little that the application programmer can do to change its characteristics to suit network conditions. As discussed above, TCP is generally not desirable for wireless networks.

Although a new transport layer protocol to replace TCP might be an answer, the inventor realized that to make the reliable transport mechanism flexible (in way that TCP is not), it would be preferable to provide the programmer with a plurality of reliable transport "personalities". While such personalities could be built into a transport layer protocol supplanting TCP, the inventor considered that it would be preferable to split the reliable transport mechanism into (1) a new application-layer protocol stack that would provide sockets to the application and use the services provided by UDP and (2) a set of personality functions that would be compiled and linked with the application, but called by the new protocol stack to perform reliable transport-related tasks. Each set of functions would constitute one discrete personality of the reliable transport mechanism.

An application programmer in writing a network application would make calls to the new protocol stack when opening a socket and sending and receiving data. The calls that the application would make would be the same regardless of the personality selected, so that the application programmer could change the desired personality by simply re-linking the application with a different set of personality functions. The application, when opening a socket, would simply provide the new protocol stack with a set of pointers to the functions for the desired personality. Alternatively, more than one set of compiled personality functions might be linked to the application and an application might then open sockets with sets of pointers to different sets of functions for communication with different server applications.

The new protocol stack, in providing reliable transport, calls the personality functions where necessary to determine parameters needed for providing reliable transport. The personality functions provide the reliable transport protocol with flexibility. For example, if no reply is received to a request, a personality function might be called and return the timeout period that the new protocol stack should wait before retransmitting a message. The function could calculate a time based by various back-off strategies, it could return a predetermined timeout period, or it could adaptively calculate a timeout period based upon current conditions.

An application programmer would be provided with the new protocol stack, the specifications for making calls to the protocol stack, and a library of sets of personality functions. The application programmer could select one or more sets of personality functions to use or could, if necessary, write a custom set of personality functions for use with the network application.

As currently implemented, the personality functions are generally used by the new protocol stack to (1) parse messages received from the transport layer, (2) add headers or trailers to requests or replies to form messages for the protocol stack to send to the transport layer, and (3) provide retransmit and reply cache timers timeout periods for use by the new protocol stack. For example, the following are done by the personality functions in one embodiment of the invention:

- setting up a connection;
- wrapping a payload (the request or reply provided to the stack by the application process) in a header to form a message to be sent to the transport layer;
- determining a time-out period for the i-th retransmission of a message;
- determining a total timeout period allowed for all retransmissions of a message; and
- parsing the messages received from the transport layer to determine the type of message (request, reply, provisional reply, or acknowledgement) for the stack.

Generally, personality functions could be used to determine any parameter of a reliable transport mechanism, handle creation and parsing of headers, and modify the data being transported. For transmissions sent between network applications that expect data to be encoded differently, personality functions could modify the payload or add additional data to the message.

Three examples of personalities are:

- a personality for over-the-air requests that provides frequent retransmissions with a hard limit on how many retransmissions are made before an error is reported;
- a personality for over-the-backhaul requests that provides exponential back-off assuming that losses could be due to congestion; and

a personality for billing requests that provides retransmissions for a very long time before giving up.

BRIEF DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the present invention will now be described, by way of example only, with reference to the attached Figures, wherein:

Figure 1 is a schematic diagram of the structure of a conventional, but minimal exemplary computer network.

Figure 2 is a schematic diagram of the protocol layer structure of a computer network implementing the inventive method.

Figure 3 is a schematic diagram of the structure of a datagram used in the network of Figure 2.

Figure 4 is a schematic diagram of a header of a message constructed by an embodiment of the inventive method.

Figures 5 – 15 are schematic diagrams showing the timing of messages passing between the client side of a protocol stack that is an embodiment of the invention and the server side of a protocol stack that is an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

Figure 2 illustrates at a high level how, in method for providing reliable transport in a computer network that is an embodiment of the invention, a request 40 is transported from a client application 50 running on a client host computer such as the client host computer 12 shown in Figure 1 to a server application 52 running on a server host computer such as the server host computer 14 shown in Figure 1. A reply by the server application 52 would be transported along the same path in the opposite direction. The boxes labeled 13 and 15 indicate the processes that take place in the client host computer 12 and the server host computer 14, respectively. The rectangular boxes, two of which are labeled with reference numerals 54 and 56, indicate the protocol layers into which the network-related processes that take place in the client host computer 12 and the server host computer 14 may be conceptually divided. Each protocol layer may be considered to provide a service to the protocol layer above it such that data submitted by an upper protocol layer to a lower protocol layer in the client host computer 12, barring losses and data corruption, will be delivered by the lower protocol layer to the upper protocol layer in the server host computer 14. Those readers skilled in the art will understand generally the nature and function of the protocol layers shown in Figure 2.

The protocol layer indicated by reference numeral 56 in Figure 2, which represents the transport layer, and the layers below it are conventional in the embodiment shown in Figure 2. For the purposes of the following discussion, the transport layer 56 is assumed to provide unreliable transport of messages sent to it by the layer indicated by reference numeral 54 in Figure 2, which represents the application layer. However, it is further assumed that the transport layer 56 (or the lower layers; it does not matter to applications that are sending request and replies over the computer network 10) provides error correction such that a message delivered to the transport layer 56 by the application layer on the client side will either be delivered uncorrupted by the transport layer 56 to the application layer 54 on the server side or not delivered at all. Further, it is assumed that between at least some client/server pairs, messages may be lost due to the nature of the physical transmission method (e.g., wireless transmission), rather than due to collisions or buffer overflow due to congestion. In the embodiment of the invention illustrated in Figure 2, the transport layer 56 implements the User Datagram Protocol ("UDP"). While not preferred, Transmission Control Protocol ("TCP") may be used instead.

As illustrated in Figure 2, in an embodiment of the invention a new protocol referred to here as the "ART Protocol" is implemented in the application layer. "ART" is an acronym for "At-most-once-signaling Reliable Transport", which is so named because, as will be understood from the discussion below, it provides a form of reliable transport for applications that minimizes the likelihood of the server application 52 processing the request 40 from the client application 50 more than once, even if the request 40 has to be retransmitted due to apparent loss.

On the client side the ART protocol is represented by a client-side ART protocol stack 58 and on the server side the ART protocol is represented by a server-side ART protocol stack 60. As is common is this art, software implementing a protocol is referred to as a "stack" and processes temporarily created by such software to handle communication with a particular remote host are referred to as "sockets". Typically stacks 58, 60 would be identical; each would create sockets for handling transmission of requests or replies to requests when called by an application, such as client application 50 and server application 52, that wishes to send requests or replies to requests to an application elsewhere in the network.

Generally, as illustrated schematically in Figures 2 and 3, the client-side ART protocol stack 58 handles a request 40 by:

wrapping the request 40 in a request message 42;

adding an ART header 43 to the request message 42; and

sending the resulting request message 42 to the transport layer 56.

As shown in Figure 4, the ART header 43 comprises a type code 46 and a segment number 48. In one implementation, the type code 46 consists of (1) a two bit binary number providing a message type and (2) a sequence number 48 that consists of a 14-bit number that distinguishes the message from other recent messages sent to the server application. In that implementation, messages may also be sent as part of the reliable transport mechanism that do not have a payload (e.g., a provisional reply) as well as messages that do have a payload, whose type is either "request" or "reply".

The reliable transport provided by the ART protocol is illustrated generally in Figures 5 –

15 by showing the exchange of messages in eleven cases. In each case time proceeds downward and messages between the client-side ART protocol stack 58 and the server-side ART protocol stack 60 are shown as downward slanting arrows indicating that each message takes a finite time to reach its destination. Each drawing also shows timer intervals: on the client side, for a retransmission timer 70; and on the server side for a reply cache timer 74. In each diagram, the retransmission timer 70 is shown as running at most twice. If the client-side ART protocol stack 58 does not obtain a reply before the second expiry, an error message will be reported to the client application and the socket will be closed. It should be noted that reporting an error after two expiries of the retransmission timer 70 would be unusual in practice, but is used here to eliminate unnecessary complexity. The handling of error messages is outside the scope of the invention.

Figure 5 schematically illustrates the timing and ordering of messages that occurs in the transmission of a request 40 from the client-side ART protocol stack 58 to the server-side ART protocol stack 60 in the ideal case in which there are neither losses of messages nor transmission or processing delays. Considering Figures 2, 3, and 4 together with Figure 5, a request 40 received from the client application 50 by an ART socket created by the client-side ART protocol stack 58 is handled as follows:

a request message 42 is formed having the request 40 as a payload and a header 43 that includes a type code 46 that indicates that the message contains the request 40 and a sequence number 48;

a retransmission timer 70 is started; and

the request message 42 handed over to the transport layer 56 for transmission to the server-side ART protocol stack 60.

The request message 42 then proceeds through the computer network 10 and is received from the transport layer 56 by the server-side ART protocol stack 60. An ART socket created by the server-side ART protocol stack 60 handles the request message 42 as follows:

the payload in the request message 42 (the request 40 itself) is handed over to the server application 52;

when a reply 72 is received from the server application process 52, a reply cache timer 74 is started;

a reply message 76 is formed having the reply 72 as a payload and a header 43 including a type code 46 that indicates that it contains a reply and a sequence number 48 that is the same as sequence number of the request message 42;

a copy of the reply message 76 is stored in a reply cache 78; and

the reply message 76 is handed over to the transport layer 56 for transmission to the client-side ART protocol stack 58.

When the reply message 76 is received by the client-side ART protocol stack 58, then:

the payload (the reply 72) is handed over to the client application 50;

the retransmission timer 70 is stopped;

an acknowledgement message 80 is formed with a header including a type code 46 indicating that it is an acknowledgement message and the sequence number of the request message 42;

and the acknowledgement message 80 is handed over to the transport layer 56 for transmission to the server-side ART protocol stack 60.

When the acknowledgement message 80 is received by the server-side ART protocol stack 60, then the cached reply message 76 in the reply cache 78 is deleted and a record 82 of the sequence number is stored. When the reply cache timer 74 expires, the stored record 82 of the sequence number 48 of the request message 42 is deleted.

Figure 6 schematically illustrates the timing and ordering of messages that occur in the transmission of a request 40 from the client-side ART protocol stack 58 to the server-side ART protocol stack 60, in the case in which there are no losses of messages, but either the request 40 is long so that it will not be provided in full to the server application 52 for some time or the server application 52 will require a long time to process the request 40 before providing a reply 72. Rather than waiting to send a reply message 76, the server-side ART protocol stack 60 sends a provisional reply 84, which as in the case of an acknowledgement message 80 may not contain a payload (unless in a particular implementation the payload is a delay period) and a header 43 including a type code 46 indicating that it is an provisional reply message and the sequence number 48 of the request 40. The client-side ART protocol stack 58 upon receiving the provisional reply message 84, resets the retransmission timer 70 to delay its expiry by a delay period 86 that could be either the payload of the provisional reply message 84 or a delay period determined by the client-side ART protocol stack 58. The time at which the retransmission timer 70 would have expired if there had been no delay is indicated by a dashed arrow labeled with reference numeral 87 and the delay period is indicated by reference numeral 86. Back at the server-side ART protocol stack 60, a reply 72 is eventually provided by the server application 52 to the server-side ART protocol stack 60. The remainder of the sequence of events is essentially the same is that discussed in relation to Figure 5. Because the retransmission timer 70 was reset, the retransmission timer 70 did not expire before the reply message 76 was received. As a result, there was no needless retransmission of the request message 42.

In the exchanges of messages illustrated in Figures 5 and 6, no messages were lost or delayed beyond their expected arrival times. Figures 7 – 15 illustrate how the ART protocol handles situations in which messages are lost or delayed. Figures 7 – 10 illustrate the exchange of messages when the request message is lost (Figure 7), the reply message is lost (Figure 8), the acknowledgement message is lost (Figure 9), and when the reply message is delayed and received after the retransmission timer expires (Figure 10). In each case the request and reply are reliably transported without reprocessing the request by the server application. Figures 11 – 13 illustrate situations in which reliable transport protocol ultimately fails and an error is reported. Figures 14 and 15 illustrate two situations in which provisional replies are lost and a second request is sent. In both cases, the request and reply are reliably transported. It should be noted that in no case illustrated in Figures 5 – 15 has a request been processed more than once. However, if an error is reported as in the cases illustrated in Figures 11-13, an error recovery procedure might result in a retransmission of the same request with a new sequence number, which may be processed again. However, the flexibility provided by the use of personalities can be used to minimize the use of an error recovery procedure unless the wireless link is totally closed down.

As will be apparent from the above discussion of Figures 5 - 15, the success and efficiency of a reliable transport mechanism depends upon an intelligent selection of the retransmission time-out periods, the total number of retransmissions before an error is reported if no reply has been received, the delay applied to the retransmission timer if a provisional reply is received, and the reply cache period. While each of these parameters could be fixed when an ART socket is created for a particular application using data passed to the ART protocol by the application, it is preferable for these parameters to be adjusted while an ART socket exists. For example, it may be best for some network applications to recalculate the retransmission time-out period each time the retransmission timer expires without a reply message having been received. The recalculation may involve consideration of recent round-trip times between the client and the server. For other applications (or other processes of the same application), it may be best to keep the retransmission time-out period constant. The inventor realized that the reliable transport mechanism could be made more flexible if the determination of these parameters were to be made by a set of functions called by the ART protocol stack, but which are supplied with the network application. An application programmer in writing or porting a network application could be provided with sets of functions suitable for a number of types of network application processes. The application programmer would not have to deal with the nuts and bolts of providing reliable transport, but would simply link a network application with a selected set of functions when creating an executable of the application, include code in the application to pass a structure containing pointers to the functions of the selected set to the ART protocol stack, and include in the application calls to the ART protocol stack. Since each set of functions provides different behavior on the part of the reliable transport mechanism, a particular set of such functions is referred here as a "personality" and the functions of a personality as "personality functions".

In addition to providing flexible and easy selection of timer parameters, the use of personalities can allow the structure of messages sent and received by an ART protocol stack to be changed by changing the personality. In particular, if a personality provides message creating and parsing functions for use by the ART protocol stack, then the structure of the ART header could be different for different personalities, without changing the application or the ART protocol stack. For example, a personality could provide a header that includes space for 32-bit sequence number. Further, by providing appropriate message creating and parsing personality functions a personality could re-encode or supplement data contained in request or reply so that the client-side of one network application could communicate with the server-side of a different network application even though the two network applications use incompatible data encoding and message structures.

More specifically, the following are the calls that a client application 50 can make to the client-side ART stack 58 in the present embodiment of the invention:

- ART_open, to open a client ART socket;
- ART_set_tos, to set the Type of Service (ToS) for the underlying IP protocol;
- ART_connect, to connect the client ART socket to a server ART socket;
- ART_request_reply, in synchronous operation, to form and send a request message 42 including the request 40 provided by the client application 50, wait for a reply message 76, and return the reply 72 to the client application 50;
- ART_send_request, in asynchronous operation, to form and send a request message 42 using the request 40 provided by the client application 50 and return the sequence number 48 of the request message 42 to the client application 50 without waiting for a corresponding reply message;
- ART_recv_reply, in asynchronous operation, to wait for a reply message 42 to the request message 40 having a specific sequence number 48 or any request message and return the reply 72 to the client application 50; and
- ART_close, to close the client ART socket.

The following are the calls that a server application 52 can make to the server-side ART stack 60 in the present embodiment of the invention:

- ART_open, to open an server ART socket;
- ART_set_tos, to set the Type of Service (ToS) for the underlying IP protocol;
- ART_recv_request, to wait for a request message and return the request to the calling server application 52;

- ART_send_provisional, to send a provisional reply message;
- ART_send_reply, to send a reply message and cache the reply message; and
- ART_close, to close a server ART socket.

A call to open an ART socket (ART_open) must include a structure that contains the pointers to the personality functions. The following are the personality functions that are called by the client-side ART protocol stack 58:

- open(), a function that creates data structures needed for a new socket, returns a starting sequence number, a minimum and maximum sequence number, header and trailer sizes, and the number of retransmissions of a request before an error is reported;
- connect(), a function that sets up the connection to the server ART socket created by the server-side ART protocol stack 60, including if necessary handshaking;
- wrap_request(), a function to form a request message 42 by adding a header 43 to a request 40 provided by the client application 50;
- retransmit_timer(), a function that returns a retransmission timer setting for the i-th retransmission when called by the client-side ART stack 58 upon sending a request message 42 to the transport layer 56 and when the retransmission timer expires before a reply message is received;
- parse(), a function to parse a message received by the client-side ART stack 58 to extract the sequence number 48 and type code 46 of message from an ART header 43, extract any payload in the message, and if the type indicates that the message is a provisional reply, then change the retransmission timer to delay its expiration;
- rtd_update(), a function that is called when a reply message is received by client-side ART stack 58 to update with the round trip time any data kept by the personality functions for use in determining timer settings;
- ack_reply(), a function to send an acknowledgement when a reply message has been received and returned to the client process; and

- close(), a function that destroys any data structures created for a specified ART socket.

The following are the personality functions that are called by the server-side ART protocol stack 60:

- open(), to create data structures needed for a new socket;
- parse(), to parse a message received by the server-side ART protocol stack 60 to extract the sequence number and type of message from the header, and extract any payload in the message;
- wrap_reply(), to form a reply message by adding a header to a reply provided by the server application or to form a provisional reply message;
- reply_cache_timer(), that returns a cache timer setting when called by the server-side ART stack upon receipt of a request message from the transport layer; and
- close(), to destroy any data structures created for a specified socket.

ART protocol stacks also include the following utility functions ("personality utility functions") that may be called by the personality functions:

- ART_get_priv(), to get a private personality state data maintained for a particular socket;
- ART_timer_request(), to maintain a private timer for a personality;
- ART_timer_cancel(), to cancel a private timer kept by a personality;
- ART_csend(), to send a message to a client-side ART stack in situations in which a message needs to be sent without receiving a call to do so from the server application;
- ART_ssend(), to send a message to a server-side ART stack in situations in which a message needs to be sent without receiving a call to do so from the client application;
- ART_defer_retransmission(), to modify the retransmission timer when the parse function has determined that a provisional reply has been received; and
- ART_reply_cache_clear(), to clear the payload of a cached reply message when the parse function has determined that an acknowledgement has been received.

As well as being embodied in a method for providing reliable transport, the invention may be embodied in the host computers in a computer network such as computer network 10 illustrated in Figure 1. In such an embodiment, the operating system 20 of the client host computer 12 includes a client-side ART protocol stack 58 for use by client applications and the operating system 30 of the server host computer 14 includes a server-side ART protocol stack 60 for use by server applications.

The following describes in more detail the actions taken by the client-side ART protocol stack 58 and the server-side ART protocol stack 60 in the current embodiment in reliably transporting a request from a client application to a server application.

Client-side ART protocol stack

```
When a command is received from a client application to open a
client socket:

        call the open personality function to have the personality
        allocate any per-connection storage it may need to keep
        data.

When a command is received from the client application to connect
the client socket to a server application:

        call the connect personality function to obtain an initial
        sequence number, a range of allowable sequence numbers, a
        maximum number of retransmissions, and header and trailer
        sizes, and to allocate resources to needed to support the
        client socket.

When a command is received from the client application to transmit
a request to the server application:

        determine the next sequence number;

        call the wrap request personality function to wrap the
        request in a request message having a header containing the
        sequence number and a message type code indicating that the
        request message contains a request;

        call the retransmit timer personality function to obtain a
        retransmit timer duration for setting the retransmit timer
        for the request;

        start the retransmit timer;

        send the request message to the transport layer for
        transmission to the server application; and

        if the command to transmit the request did not block inside
        the ART stack until a reply was received or all retransmit
        times expired, then return the sequence number to the client
        application.

When a message is received from the transport layer:
```

call the parse personality function to obtain the message type, sequence number of the message, and the payload, if any, and if the message type and sequence number indicates that the message is a provisional reply to the request, to modify the retransmit timer to delay its expiration;

if the message type returned by the parse personality function indicates that the message contained a reply as a payload, then

if the sequence number returned matches an outstanding request, then

if the command to transmit the request specified that the reply be returned to the client application upon its receipt from the transport layer, or the client application, since sending the command to transmit the request, has sent a command to return the reply upon its receipt from the transport layer, then return the reply to the client application, but otherwise store the reply.

When a command is received from the client application to return a reply that is received to the request and the reply has been received and stored:

return the reply to the client application.

After the reply is returned to the client application:

call the acknowledgement reply personality function, which may send a message to the transport layer for transmission to the server application acknowledging the return of the reply to the client application, the message containing the sequence number and an indication that the message is an acknowledgement to the reply message.

If no message having a header containing the sequence number and a type code indicating that the message contains a reply has been received before the retransmit timer has expired, then repeatedly:

call the retransmit timer personality function to obtain a new retransmit timer setting for setting the retransmit timer;

set and start the retransmit timer; and

send the request message to the transport layer for transmission to the server application,

until the retransmit timer has expired the maximum number of times or until a message having a header containing the sequence number and a type code indicating that the message contains a reply is received from the transport layer,

but if the retransmit timer has expired the maximum number of times and no such message has been received, then:

report a transmission error to the client application and destroying any subsequent messages having a header containing the sequence number until the sequence number is assigned to a new request message.

When a command is received from the client application to close the client socket,

call the close personality function to free up any resources allocated to support the client socket.

Server-side ART protocol stack

When a command is received from the server application to open a server socket:

call the open personality function to allocate resources needed to support the server socket.

When a message is received from the client application:

call the parse personality function to obtain the message type, the sequence number of the message, and the request, if the message type indicates that the message contained a request, and then

if a request is returned by the parse personality function, return the request to the server application if a command was previously received from the server application to receive a request from the client application, but otherwise store the request for future retrieval by the server application.

If, after delivery of the request to the server application, a command is received from the server application to send a provisional reply, then:

call the wrap personality function to form a provisional reply message having a sequence number and a message type code indicating that the message is a provisional reply to the request message, and then

send the provisional reply message to the transport layer for transmission to the client application.

When a command is received from the server application to transmit a reply to the client application:

call the wrap reply personality function to wrap the reply in a header containing the sequence number and a message type code indicating that the request message contains a reply,

call the reply cache timer personality function to obtain a reply cache timer setting for setting a reply cache timer that counts down from the setting,

start the reply cache timer,

send the resulting message to the transport layer for transmission to the client application, and

cache a copy of the reply message.

When a message is received from the client application subsequent to the first message containing the request:

call the parse personality function to

```
                obtain the message type and sequence number of the
                message, and

                to delete the payload from the cached reply message if
                the message contains an indication that the message is
                an acknowledgement to a cached reply message; and

          if the message type indicates that the message contains a
          request and a cached reply message has the same sequence
          number, then

                resend the cached reply message to the transport layer
                for transmission to the client application.

    When the reply cache timer expires:

          delete the cached reply message.

    When a command is received from the server application to close
    the server socket:

          call the close personality function to free up any resources
          allocated by the personality functions.
```

The above-described embodiments of the invention is intended to be examples of the present invention, and alterations and modifications may be effected thereto by those of skill in the art, without departing from the scope of the invention which is defined solely by the claims appended hereto.

Claims:

1. A method for reliably transmitting, over a computer network from a client application to a server application, a request wrapped in a request message, in which method the client application and the server application respectively provide a client protocol stack and a server protocol stack with functions including:

a function that the client protocol stack may call to obtain parameter values that govern the reliable transport of the request message to the server protocol stack for delivery to a server application; and

a function that the server protocol stack may call to obtain parameter values that govern the reliable transport of a reply wrapped in a reply message from the server protocol stack to the client protocol stack for delivery to a client application.

2. The method of claim 1, wherein the functions include:

a function for determining a time-out period for an i-th transmission of the request message and a total allowed number of transmissions of the request message to be made before an error is reported; and

a function for determining a time-out period for a reply cache timer.

3. The method of claim 2, wherein the functions include a function for wrapping the request in the request message and a function for wrapping the reply in the reply message.

4. The method of claim 3, wherein:

the function for wrapping the request in the request message adds a header to the request message that includes an identifying sequence number assigned by the client protocol stack and a type code identifying the type of the request message as a request; and

the function for wrapping the reply in the reply message adds a header to the reply message that includes the identifying sequence number assigned by the client protocol stack to the corresponding request and a type code identifying the type of the reply message as a reply

5. The method of claim 4, wherein the functions include a function that returns a

starting sequence number and range of allowable sequence numbers for request messages.

6. The method of claim 5, wherein the functions include a function that parses a message and returns the type and sequence number of that message, and, if the message includes a payload, the payload.

7. The method of claim 6, wherein the client protocol stack retransmits the request message if a message received from the server protocol stack contains the sequence number of the request message and a type code that indicates that it is a reply is not received before the time-out period for i-th transmission has elapsed since the request message was transmitted for the i-th time and the number of times that the request message has been transmitted is less than the total allowed number of transmissions.

8. The method of claim 7, wherein, if a message received by the client protocol stack from the server protocol stack contains the sequence number of the request message and a type code that indicates that it is a provisional reply message to the request message, then the current time-out period for retransmission is modified to delay its expiration.

9. The method of claim 8, wherein, if the message type of a message received from the client protocol stack indicates that the message contains a request, delivering the request to the server application.

10. The method of claim 9, wherein, if, after delivery of the request to the server application, the server application wishes to send a provisional reply, then sending a message to the client protocol stack containing the sequence number and a message type code indicating that the message is a provisional reply to the request message.

11. The method of claim 9, wherein:

when a reply message is sent by the server protocol stack to the client protocol stack, starting the reply cache timer, caching the reply message in a reply cache, and destroying the cached reply message upon the expiration of the reply cache time-out period;

if the message type of a message received by the server protocol stack from the client protocol stack indicates that the message contains a request and a cached reply message has the same sequence number, then resending the cached reply message to the client protocol stack;

if the message type of a message received by the server protocol stack from the client protocol stack indicates that the message is an acknowledgement to a cached reply message, then retaining only the sequence number in the cached reply message in the reply cache; and

when the reply cache timer expires, deleting the cached reply message.

12. A method for reliably transmitting a request submitted by a client application to a client protocol stack over a computer network wrapped in a request message to a server protocol stack for delivery to a server application, in which method the client application provides the client protocol stack with a set of functions including at least one function that the client protocol stack may call to obtain parameter values that govern the reliable transport of the request message.

13. The method of claim 12, wherein the set of functions includes a function for determining a time-out period for an i-th transmission of the request message and a total allowed number of transmissions of the request message to be made before an error is reported.

14. The method of claim 12, wherein the set of functions includes a function that the client protocol stack may call to wrap the request in the request message.

15. The method of claim 14, wherein the function that the client protocol stack may call to wrap the request in the request message adds a header to the request message that includes an identifying sequence number assigned by the client protocol stack and a type code identifying the type of the request message as a request.

16. The method of claim 15, wherein the set of functions includes a function for determining a time-out period for an i-th transmission of the request message and a total allowed number of transmissions of the request message to be made before an error is reported.

17. The method of claim 15, wherein the set of functions includes a function that the client protocol stack may call to obtain a starting sequence number and range of allowable sequence numbers for request messages.

18. The method of claim 17, wherein the set of functions includes a function that the client protocol stack may call to parse a message received from the server protocol stack and that returns the type and sequence number of that message to the client protocol stack.

19. The method of claim 18, wherein the client protocol stack retransmits the request message if a message received from the server protocol stack contains the sequence number of the request message and a type code that indicates that it is a reply is not received before the time-out period for i-th transmission has elapsed since the request message was transmitted for the i-th time and the number of times that the request message has been transmitted is less than the total allowed number of transmissions, but if the type code and sequence number indicate that the message is a provisional reply message to the request message, then modifying the current time-out period to delay its expiration.

20. The method of claim 19, wherein the set of functions includes a function for determining a time-out period for an i-th transmission of the request message and a total allowed number of transmissions of the request message to be made before an error is reported.

21. A method for reliably transmitting over a computer network a request wrapped in a request message, in which method a server application provides a server protocol stack with a set of functions including at least one function that the server protocol stack calls to obtain parameter values that govern the reliable transport of a reply message from a server protocol stack to a client protocol stack.

22. The method of claim 21, wherein the set of functions includes a function for determining a time-out period for a reply cache timer.

23. The method of claim 22, wherein the set of functions includes a function that the server protocol stack calls to wrap the reply in the reply message.

24. The method of claim 23, wherein the function that the server protocol stack calls to wrap the reply in the reply message adds a header to the reply message that includes the identifying sequence number assigned by the client protocol stack to the corresponding request and a type code identifying the type of the reply message as a reply.

25. The method of claim 24 wherein the set of functions includes a function that the server protocol stack may call to parse a message received from the client protocol stack that returns the type and sequence number of that message.

26. The method of claim 25, wherein, if the message type of a message received from the

client protocol stack indicates that the message contains a request, delivering the request to the server application.

27. The method of claim 26, wherein, if, after delivery of the request to the server application, the server application wishes to send a provisional reply, then sending a message to the client protocol stack containing the sequence number and a message type code indicating that the message is a provisional reply to the request message.

28. The method of claim 27, wherein, when a reply message is sent to the client protocol stack, starting the reply cache timer, caching the reply message in a reply cache, and destroying the cached reply message upon the expiration of the time-out period.

29. The method of claim 28, wherein, if the message type of a message received from the client protocol stack indicates that the message contains a request and a cached reply message has the same sequence number, then resending the cached reply message to the client protocol stack;

30. The method of claim 29, wherein, if the message type of a message received from the client protocol stack indicates that the message is an acknowledgement to a cached reply message, then retaining only the sequence number in the cached reply message in the reply cache; and

31. The method of claim 30, wherein when the reply cache timer expires, deleting the cached reply message.

32. A system for reliably transmitting a request wrapped in a request message over a computer network to a server protocol stack in a server host computer, comprising a client host computer having a client protocol stack for receiving the request from a client application, wrapping the request in a request message, and providing reliable transport of the request message, the client protocol stack when so doing calling a set of functions external to the client protocol stack and provided by the client application, the set of functions providing reliable transport-related services to the client protocol stack than may pre-selected to meet the requirement of the client application.

33. The system of claim 32, wherein the set of functions includes a function for

determining a time-out period for an i-th retransmission of the request message and a total allowed number of transmissions of the request message to be made before an error is reported.

34. The system of claim 33, wherein the set of functions includes a function that the client protocol stack may call to wrap the request in the request message.

35. The system of claim 34, wherein the function that the client protocol stack may call to wrap the request in the request message adds a header to the request message including an identifying sequence number assigned by the client protocol stack and a type code identifying the type of the request message as a request.

36. The system of claim 35, wherein the set of functions includes a function that the client protocol stack may call to obtain a starting sequence number and range of allowable sequence numbers for request messages.

37. The system of claim 36, wherein the set of functions includes a function that the client protocol stack may call to parse a message received from the server protocol stack and obtain the type and sequence number of that message.

38. The system of claim 37, wherein the client protocol stack retransmits the request message if a message received from the server protocol stack contains the sequence number of the request message and a type code that indicates that it is a reply is not received before the time-out period for i-th transmission has elapsed since the request message was transmitted for the i-th time, and the number of times that the request message has been transmitted is less than the total allowed number of transmissions, but if a message received from the server protocol stack contains the sequence number of the request message and a type code that indicates that the message is a provisional reply message to the request message, then the current time-out period is changed to delay its expiration.

39. A server host computer for use in providing reliable transport over a computer network, the server host computer having a server protocol stack for receiving a request message transmitted from a client host computer, providing a request wrapped in the request message to a server application running on the server host computer, wrapping a reply received from the server application in a reply message, and transmitting the reply message back to the client host computer, the server protocol stack when so doing calling a set of functions external to the

server protocol stack and provided by the server application, the set of functions providing reliable transport-related services to the server protocol stack than may pre-selected to meet the requirements of the server application.

40. The server host computer of claim 39, wherein the set of functions includes a function for determining a time-out period for a reply cache timer.

41. The server host computer of claim 40, wherein the set of functions includes a function that the server protocol stack may call to wrap the reply in the reply message.

42. The server host computer of claim 41, wherein the function that the server protocol stack may call to wrap the reply in the reply message adds a header to the reply message including the identifying sequence number assigned by the client protocol stack to the corresponding request and a type code identifying the type of the reply message as a reply.

43. The server host computer of claim 42 wherein the set of functions includes a function that the server protocol stack may call to parse a message received from the client protocol stack and obtain the type and sequence number of that message.

44. The server host computer of claim 43, wherein, if the message type of a message received from the client protocol stack indicates that the message contains a request, then the request is delivered to the server application.

45. The server host computer of claim 44, wherein, if after delivery of the request to the server application the server application wishes to send a provisional reply, then a message is sent to the client protocol stack containing the sequence number and a message type code indicating that the message is a provisional reply to the request message.

46. The server host computer of claim 45, wherein, when a reply message is sent to the client protocol stack, then the reply cache timer is started, the reply message is cached in a reply cache, and the cached reply message destroyed upon the expiration of the time-out period.

47. The server host computer of claim 46, wherein, if the message type of a message received from the client protocol stack indicates that the message contains a request and a cached reply message has the same sequence number, then the cached reply message is resent to the client protocol stack;

48. The server host computer of claim 47, wherein, if the message type of a message received from the client protocol stack indicates that the message is an acknowledgement to a cached reply message, then only the sequence number in the cached reply message is retained in the reply cache; and

49. The server host computer of claim 48, wherein when the reply cache timer expires, then the cached reply message is deleted.

50. A method for reliably transmitting over a network a request from a client application running on a client system to a server application running on a server system, the method comprising, in the client system:

(a) when a command is received from the client application to open a client socket,

receiving from the client application a set of pointers to a set of personality functions provided by the client application, which include

an open function,

a connect function,

a wrap request function,

a retransmit timer function,

a parse function,

a retransmit date update function,

an acknowledgement reply function, and

a close function, and

calling the open function to obtain an initial sequence number, a range of allowable sequence numbers, a maximum number of retransmissions, and header and trailer sizes, and to allocate resources to needed to support the client socket;

(b) when a command is received from the client application to connect the client socket to the server application, calling the connect function to open a connection to the server application;

(c) when a command is received from the client application to transmit a request to the server application,

determining the next sequence number,

calling the wrap request function to wrap the request in a request message having a header containing the sequence number and a message type code indicating that the request message contains a request,

calling the retransmit timer function to obtain a retransmit timer setting for setting a retransmit timer that counts down from the retransmit timer setting,

starting the retransmit timer at the retransmit timer setting,

sending the request message to the transport layer for transmission to the server application, and

if the command to transmit the request asked that sequence number assigned to the message to be returned, then returning the sequence number to the client application;

(d) when a message is received from the transport layer,

calling the parse function to obtain the message type and sequence number of the message, and if the message type and sequence number indicate that the message is a provisional reply to the request, then modifying the retransmit timer to delay its expiration, and

if the message type and sequence number indicate that the message contains a reply to the request,

returning the reply to the client application if the command to transmit the request specified that the reply be returned to the client application upon its receipt from the transport layer, or

the client application, since sending the command to transmit the request, has sent a command to return the reply upon its receipt from the transport layer,

but otherwise storing the reply;

(e) when a command is received from the client application to return the reply to the request and the reply has been received and stored,

returning the reply to the client application;

(f) when the reply is returned to the client application,

calling the acknowledgement reply function to send a message to the transport layer for transmission to the server application acknowledging the return of the reply to the client application, the message containing the sequence number and an indication that the message is an acknowledgement to the reply message;

(g) if no message having a header containing the sequence number and a type code indicating that the message contains a reply has been received before the retransmit timer has expired, then repeatedly

calling the retransmit timer function to obtain a new retransmit timer setting for setting the retransmit timer,

setting and starting the retransmit timer, and

sending the request message to the transport layer for transmission to the server application,

until the retransmit timer has expired the maximum number of times or until a message having a header containing the sequence number and a type code indicating that the message contains a reply is received from the transport layer,

but if the retransmit timer has expired the maximum number of times and no such message has been received, then

reporting a transmission error to the client application and destroying any subsequent messages having a header containing the sequence number until the sequence number is assigned to a new request message;

(h) when a command is received from the client application to close the client socket,

calling the close function to free up any resources allocated to support the client socket;

and in the server system:

(i) when a command is received from the server application to open a server socket,

receiving from the server application a set of pointers to a set of personality functions, which include

the open function,

the parse function,

a wrap reply function,

a reply cache timer function, and

the close function, and

calling the open function to allocate resources needed to support the server socket;

(j) when a command is received from the server application to receive a request from the client application and then a message is received from the client application,

calling the parse function to obtain the message type and sequence number of the message and, if the message type indicates that the message contains a request, the request, and then returning the request to the server application process;

(k) if, after delivery of the request to the server application, a command is received from the server application to send a provisional reply, then

sending a message to the transport layer for transmission to the client application containing the sequence number and a message type code indicating that the message is a provisional reply to the request message;

(l) when a command is received from the server application to transmit a reply to the client application,

calling the wrap reply function to wrap the reply in a header containing the sequence number and a message type code indicating that the request message contains a reply,

calling the reply cache timer function to obtain a reply cache timer setting for setting a reply cache timer that counts down from the setting,

starting the reply cache timer,

sending the resulting message to the transport layer for transmission to the client application, and

caching the reply message;

(m) when a further message is received from the client application,

calling the parse function to obtain the message type and sequence number of the message and

if the message type indicates that the message contains a request and a cached reply message has the same sequence number, then resending the cached reply message to the transport layer for transmission to the client application, and

if a message contains an indication that the message is an acknowledgement to a cached reply message, then

deleting everything except the sequence number from the cached reply message;

(n) when the reply cache timer expires, then

deleting the cached reply message; and

(o) when a command is received from the server application to close the server socket,

calling the close function to free up any resources allocated by the personality functions

.

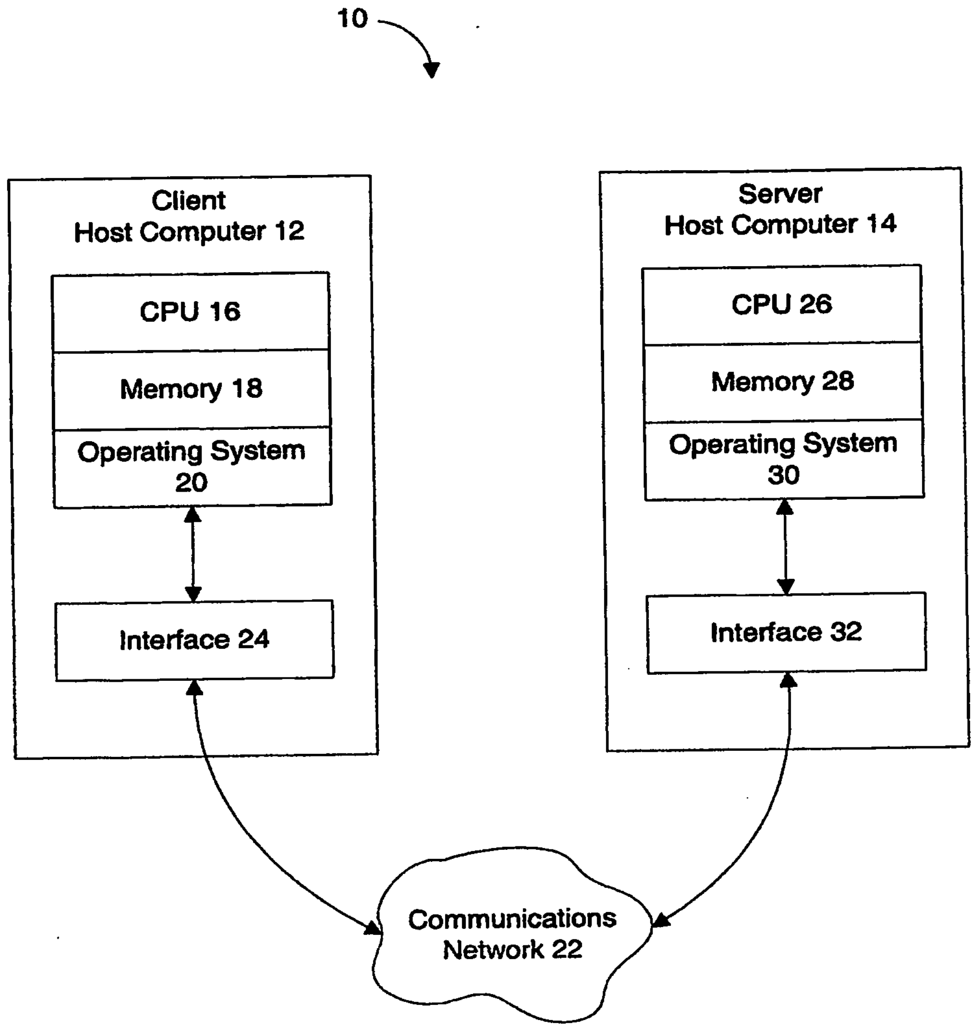
10
Client Host Computer 12
CPU 16
Memory 18
Operating System 20
Interface 24
Server Host Computer 14
CPU 26
Memory 28
Operating System 30
Interface 32
Communications Network 22

Figure 1

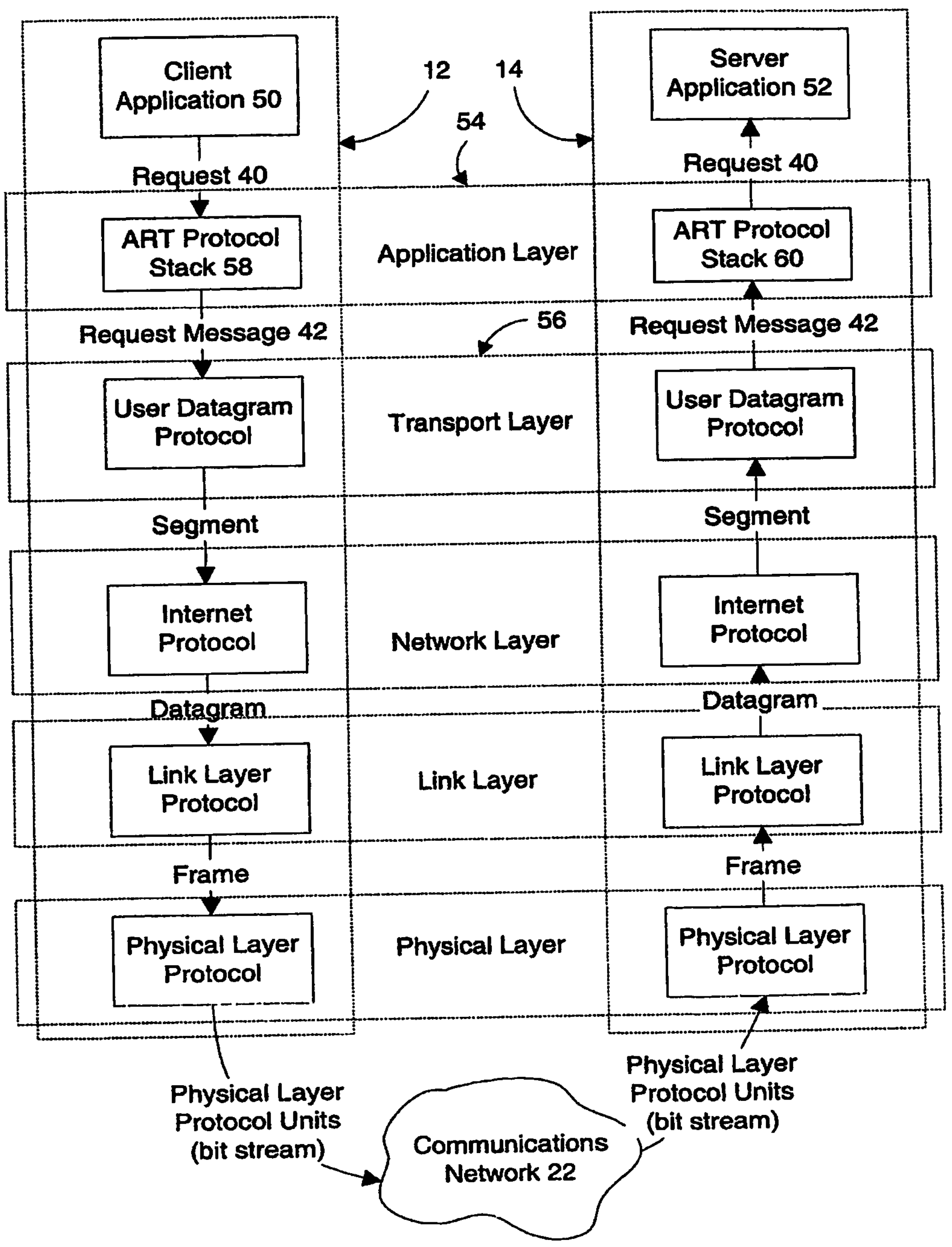
Client Application 50
12
14
54
Server Application 52
Request 40
Request 40
ART Protocol Stack 58
Application Layer
ART Protocol Stack 60
Request Message 42
56
Request Message 42
User Datagram Protocol
Transport Layer
User Datagram Protocol
Segment
Segment
Internet Protocol
Network Layer
Internet Protocol
Datagram
Datagram
Link Layer Protocol
Link Layer
Link Layer Protocol
Frame
Frame
Physical Layer Protocol
Physical Layer
Physical Layer Protocol
Physical Layer Protocol Units (bit stream)
Physical Layer Protocol Units (bit stream)
Communications Network 22

Figure 2

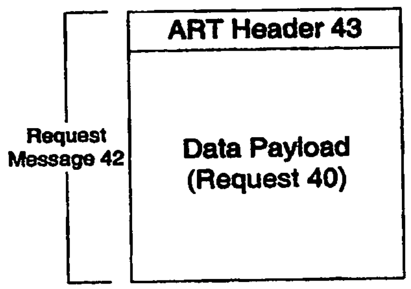

Figure 3

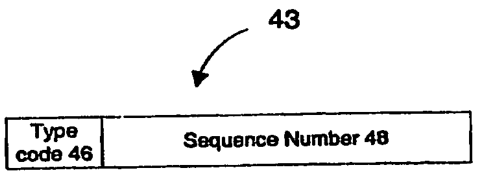

Figure 4

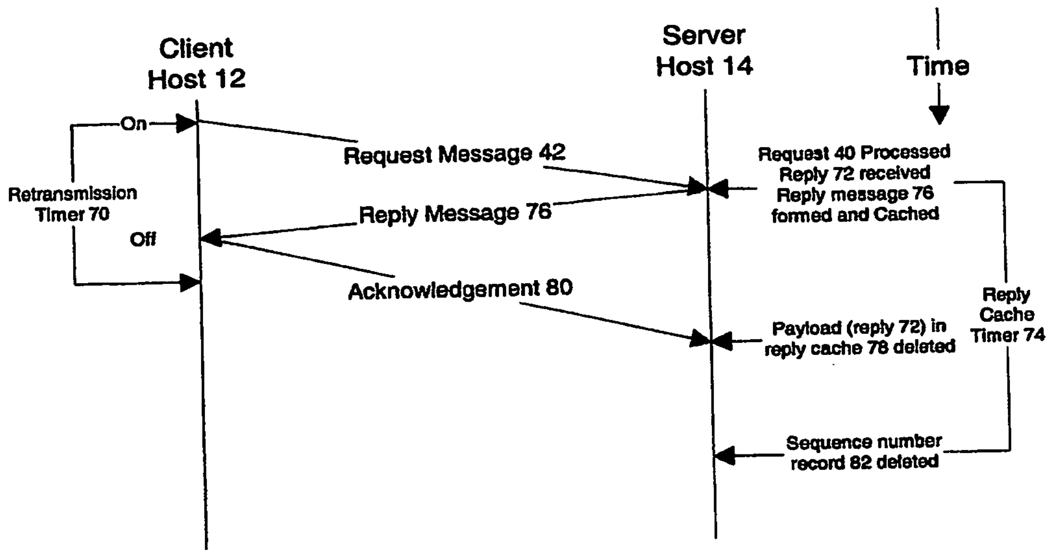

Figure 5

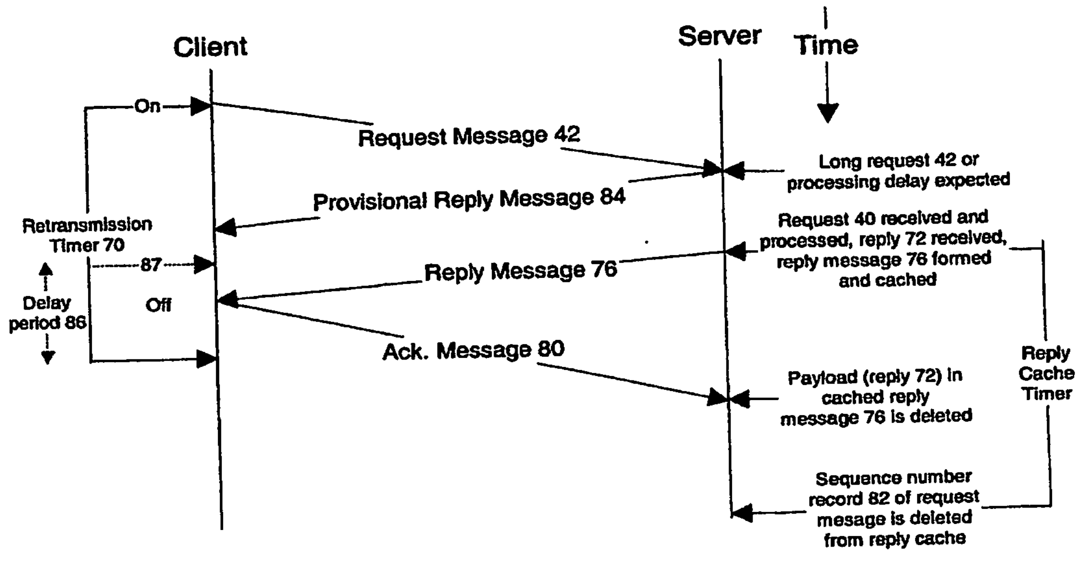

Figure 6

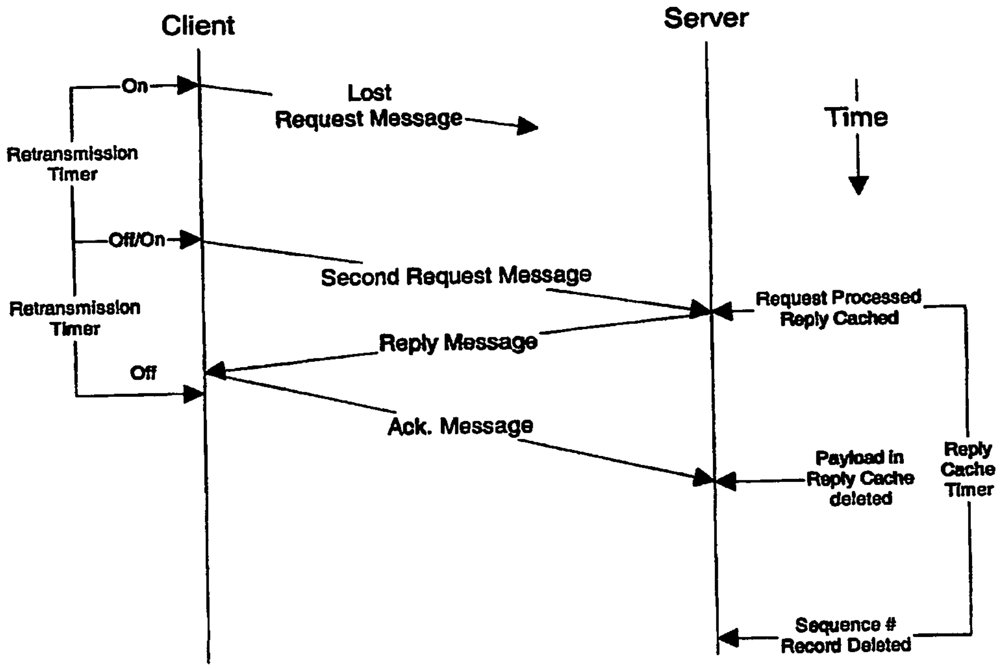

Figure 7

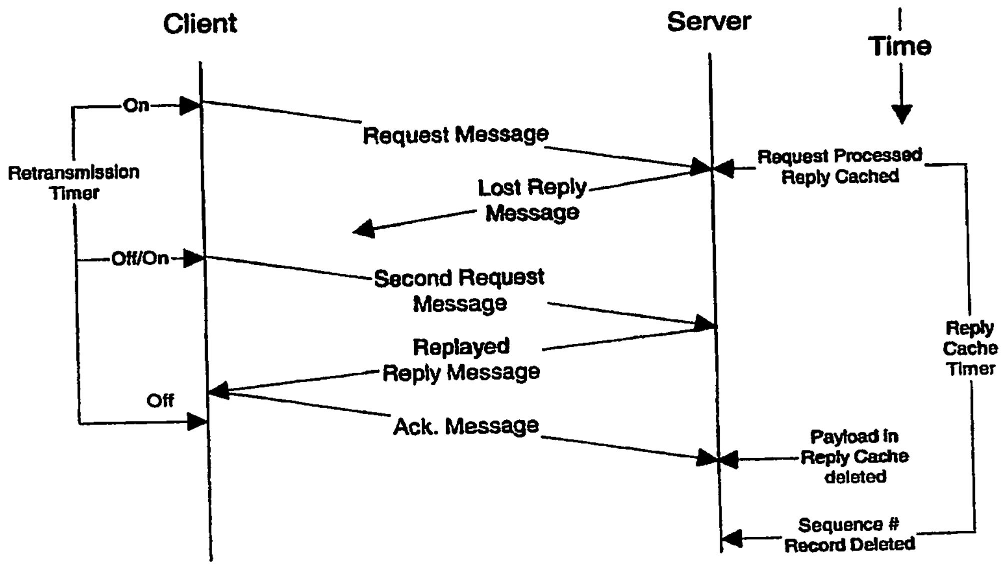

Figure 8

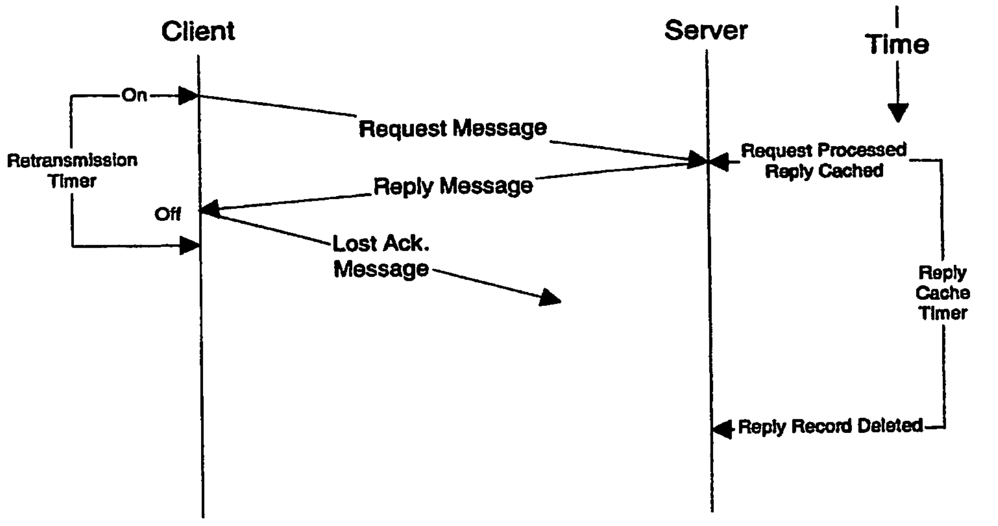

Figure 9

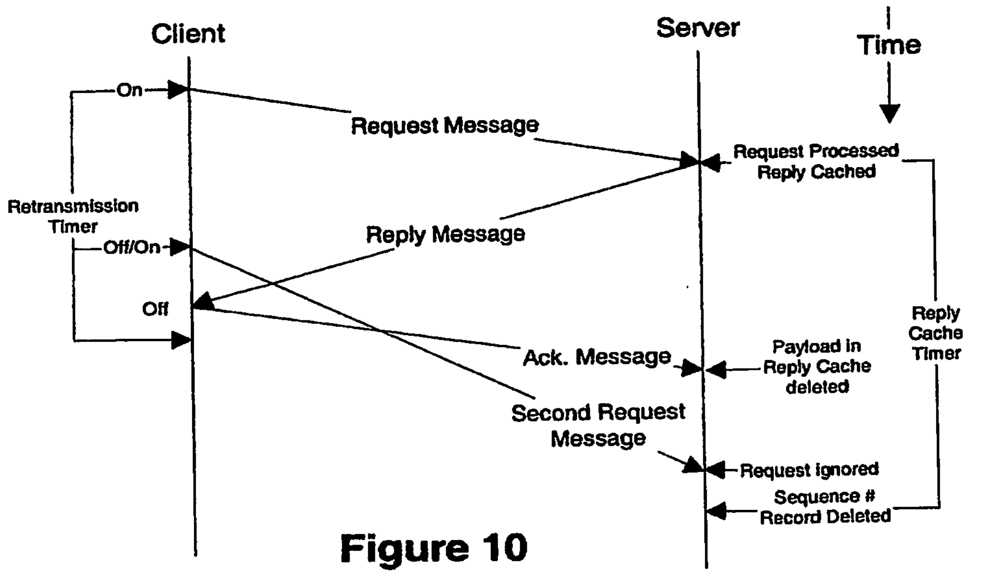

Figure 10

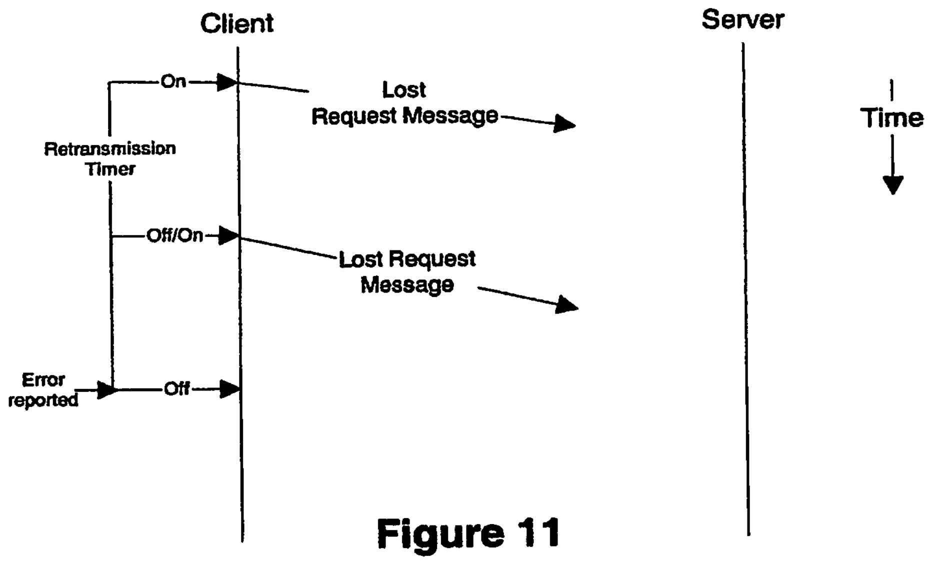

Figure 11

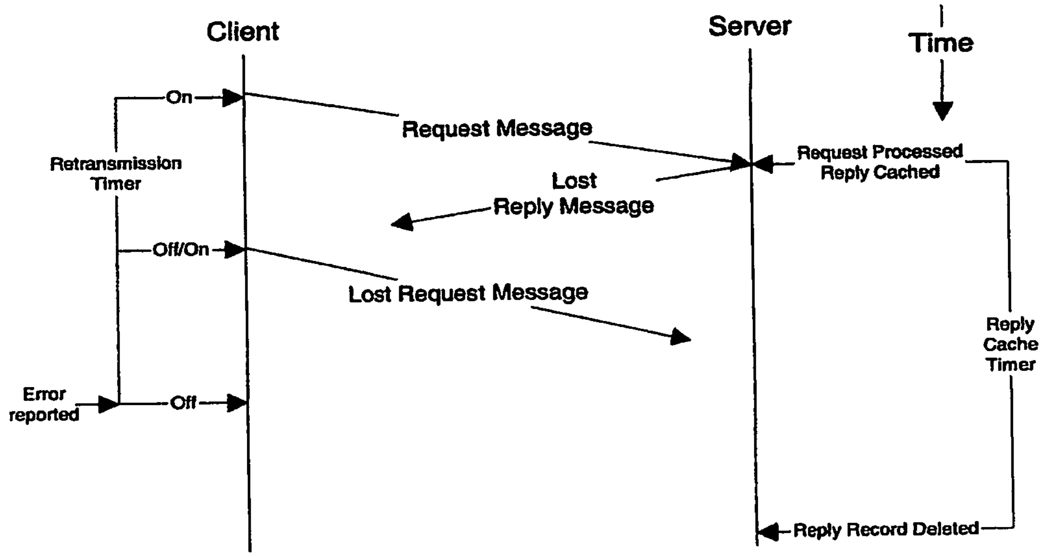

Figure 12

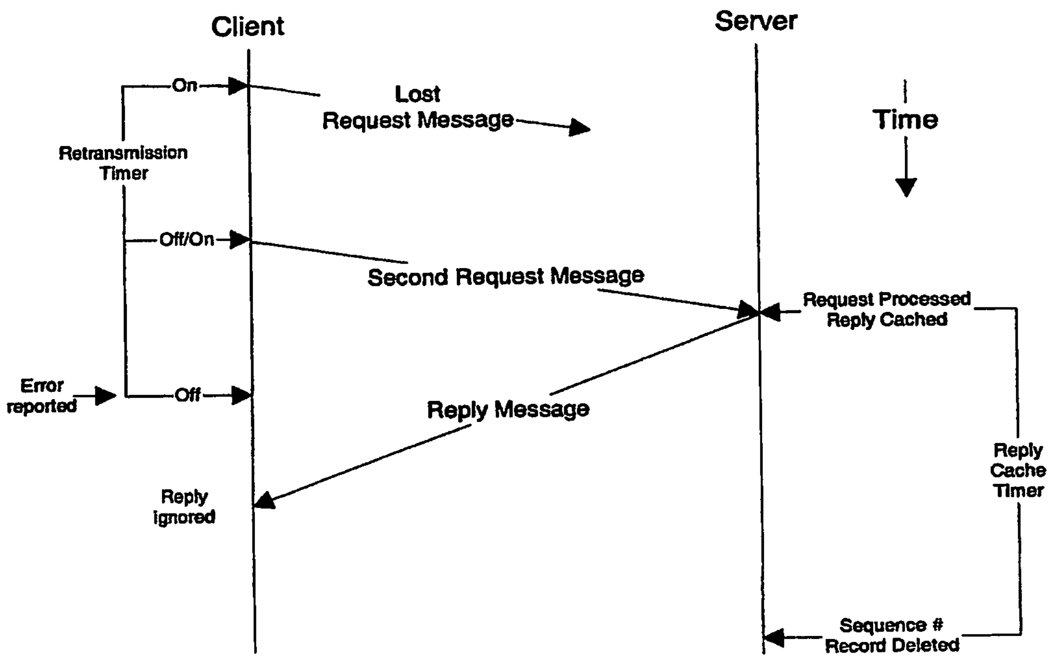
Client
Server
Time
On
Lost
Request Message
Retransmission
Timer
Off/On
Second Request Message
Request Processed
Reply Cached
Error
reported
Off
Reply Message
Reply
Cache
Timer
Reply
ignored
Sequence #
Record Deleted

Figure 13

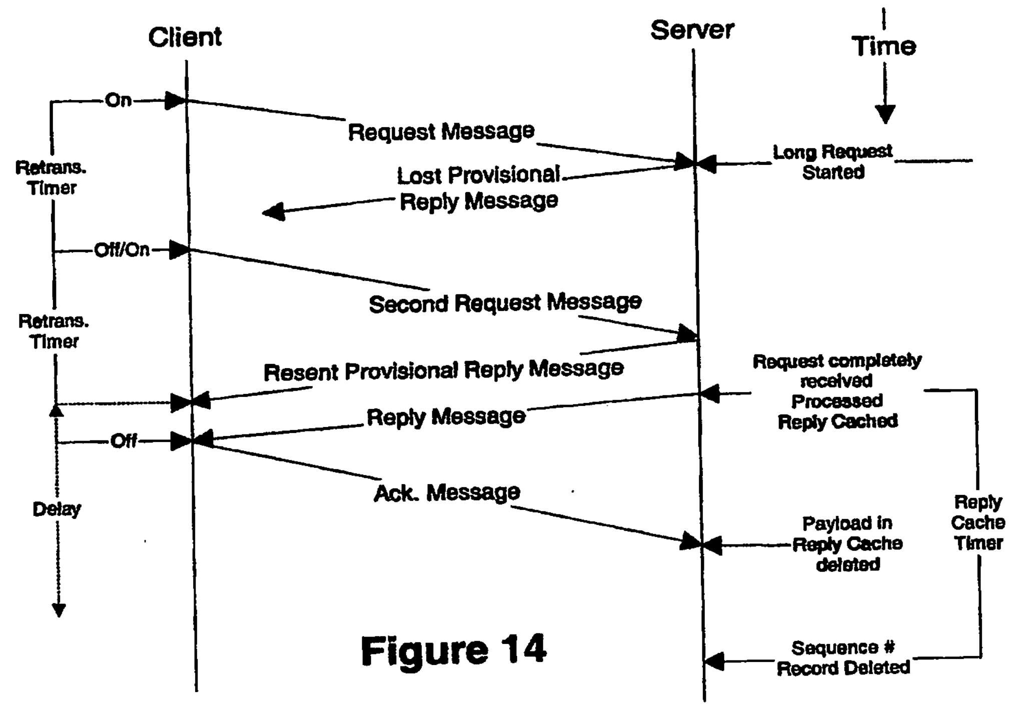
Client
Server
Time
On
Request Message
Long Request Started
Retrans. Timer
Lost Provisional Reply Message
Off/On
Second Request Message
Retrans. Timer
Request completely received Processed Reply Cached
Resent Provisional Reply Message
Reply Message
Off
Ack. Message
Delay
Payload in Reply Cache deleted
Reply Cache Timer
Sequence # Record Deleted

Figure 14

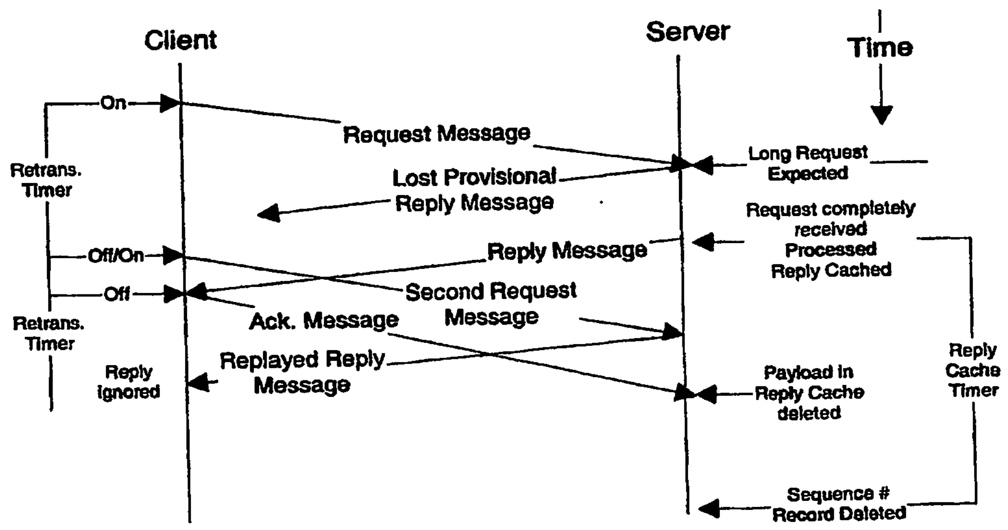

Figure 15